Linguagem Imperativa Modelada em Haskell

Lucas Iankowski e Bruno Wide

[1]Faculdade de Informática - Pontifícia Universidade Católica Rio Grande do Sul

[2]Ciência da Computação

Resumo. *Neste artigo temos como objetivo detalhar a implementação do algoritmo desenvolvido em linguagem Haskell, onde definindo uma sintaxe e semântica, criamos comandos que modelam uma nova linguagem imperativa.*

1. Introdução

A linguagem Haskell nos permite, de maneira menos complexa, criar uma nova sintaxe e semântica para operações lógica-matemáticas, e através de estruturas da linguagem Haskell vistas em aula, implementaremos uma nova linguagem imperativa.

Além disso, vale mencionar que a implementação deste problema encerra a primeira metade do conteúdo de Programação Funcional, sabendo que foram utilizadas ao longo do programa apenas as estruturas vistas na primeira metade do semestre.

2. Implementação

A implementação do problema proposto foi subdividida em 4 partes.

Memory, onde armazenamos e manipulamos as variáveis criadas, Primitives, onde definimos os tipos primitivos, Commands, onde definimos o formato de chamada das funções criadas e Math, que é o main executável do programa e onde recebe as instruções operacionais (inputs).

2.1. Memory

A memória possui 3 (estados/ações) que podem ser executados, cada posição da memória guarda uma tupla (String, Inteiro), referente a sua variável armazenada.

Initial - Onde inicializamos a memória: Memory[NULL].

Value - Onde recebemos a memória, um nome de variável e retorna uma variável: Busca na memória o valor de uma variável, caso não exista retorna nulo.

Update - Onde recebemos a memória, um nome de variável, um inteiro e retornamos a memória: Atualiza o estado da memória, e se necessário o valor da variável.

2.2. Primitives

Aqui, como o nome sugere, criamos os tipos primitivos, definimos a sintaxe e o formato das expressões.

Uma expressão aritmética recebe dois inteiros com ”+”ou ”*”entre eles. Já uma expressão booleana pode ser usada em operações de maior, menor, and, or e not, na qual maior, menor, and e or operam com duas expressões e not com apenas uma expressão.

As operações possiveis na nossa linguagem são:

Atribuição, onde se recebe o nome e valor da variável.

Seq, pra executar duas expressões ao mesmo tempo.

If, recebemos uma expressão booleana e duas expressões, se a expressão booleana for verdadeira, executa a primeira expressão, senão a segunda.

Pré-loop, onde primeiro é recebido a expressão booleana que é executada enquanto for verdadeira e depois uma expressão.

Pós-loop, onde primeiro se recebe a expressão e depois é testada a expressão booleana.

2.3. Commands

Aqui definimos os comandos de cálculo:

A função calculateBool recebe uma expressão booleana e retorna um booleano. Já a função calculate é pra cálculos aritméticos, então recebe uma expressão aritmética e retorna um inteiro.

A função calc recebe uma expressão e a memória, retorna a memória, então:

Se receber NULL, retorna a memória sem alterações.

Se receber uma atribuição então dá update na memória com o novo valor da variável.

Se receber duas operações então calcula uma de cada vez e armazena na memória.

Se receber um condicional então se a expressão booleana for verdadeira calcula cmd1, senão cmd2, por fim armazena na memória.

Se receber um pré-loop, então se a expressão condicional for verdadeira, calcula cmd, senão retorna o estado de memória sem alteração.

Se receber um pós-loop, então calcula a expressão, e se a expressão booleana do condicional for verdadeira, então repete o cálculo, senão retorna a memória como está.

2.4. Math

Esta é a parte de mais alto nível da nossa implementação, onde já temos definido tudo em relação a sintaxe e semântica. Aqui apenas operamos as instruções recebidas por input.

A função sum recebe dois inteiros e retorna um inteiro, chama a função calculate e armazena na memória o resultado.

A função mult recebe dois inteiros e retorna um valor, enquanto o valor armazenado na variável ”y”for maior que zero, então é somado executado uma atribuição ”x = x + x”. e decrementa ”y”.

3. Conclusão

Com a realização deste artigo e implementação, pode-se concluir que a linguagem Haskell realmente é muito poderosa quando se trata de manipulação de estruturas e o seu controle. É uma ferramenta muito interessante e que não é viável em linguagens não-funcionais.

4. Referências

Para realizar a implementação foi utilizado como fonte de conhecimento o livro ”The Craft of Functional Programming”, de Simon Thompson e alguns fóruns para resolver problemas bem específicos.